1988-10

A Prototyping Language for Real-Time Software

Luqi

IEEE Transactions on Software Engineering, Vol. 14, No. 10, October 1988
http://hdl.handle.net/10945/39162

A Prototyping Language for Real-Time Software

LUQI, VALDIS BERZINS, MEMBER, IEEE, AND RAYMOND T. YEH, FELLOW, IEEE

Abstract—**PSDL is a language for describing prototypes of real-time software systems. It is most useful for requirements analysis, feasibility studies, and the design of large embedded systems. PSDL has facilities for recording and enforcing timing constraints, and for modeling the control aspects of real-time systems using nonprocedural control constraints, operator abstractions, and data abstractions. The language has been designed for use with an associated prototyping methodology. PSDL prototypes are executable if supported by a software base containing reusable software components in an underlying programming language (e.g., Ada).**

Index Terms—**Abstractions, Ada, embedded systems, prototyping, real-time systems, reusable components, specification language.**

I. INTRODUCTION

THE rapidly growing demand for software has shifted towards larger systems and higher quality software, to a point beyond the reach of current software development methods. A jump in software technology is needed to improve programming productivity and the reliability of the software product. Rapid prototyping is one of the most promising methods proposed to reach this goal. This paper presents a prototype system description language (PSDL) which supports rapid prototyping based on abstractions and reusable software components. PSDL is especially well suited for large real-time systems, and should be useful for prototyping typical Ada applications.

PSDL is well suited for use with Ada. Libraries of reusable Ada components are being assembled for practical use, and should continue to grow because Ada is required by many large contracts and is used in an increasing number of software systems. The use of the same programming language in the software base and in the final implementation encourages the transfer of reusable components from the prototype to the production software in cases where further details and improvements are not needed. Ada is convenient for implementing PSDL because the mechanisms of Ada support the features of PSDL and because it is easier to interface to reusable components if they are in the same language as the PSDL execution support system.

Manuscript received June 5, 1986; revised October 2, 1986.

Luqi was with the Department of Computer Science, University of Minnesota, Minneapolis, MN 55455. She is now with the Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943.

V. Berzins is with the Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, and the Department of Computer Science, Naval Postgraduate School, Monterey, CA 93943.

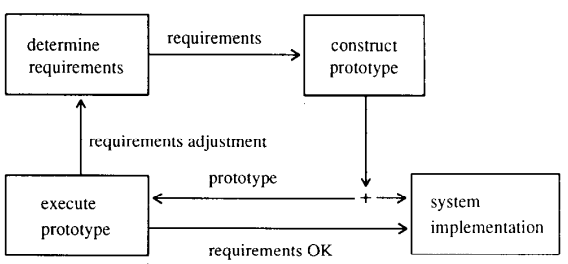
R. Yeh is with International Software Systems, Inc., 12710 Research Blvd., Austin, TX 78759.

IEEE Log Number 8823081.

A. Conceptual Framework

A prototype is an executable model or a pilot version of the intended system. A prototype is usually a partial representation of the intended system, used as an aid in analysis and design rather than as production software. The rapid construction activity leading to such a prototype is called rapid prototyping. Rapid prototyping has been found to be an effective technique for clarifying requirements and eliminating the large amount of wasted effort currently spent on developing software to meet incorrect or inappropriate requirements [29] in traditional software life cycles. Lack of agreement on the requirements as specified by the customer and as analyzed by the designer causes inconsistencies between the delivered system and customer expectations, leading to expensive rebuilding. This problem is especially acute for large systems and systems with real time constraints because the requirements for such systems are complicated and difficult to understand.

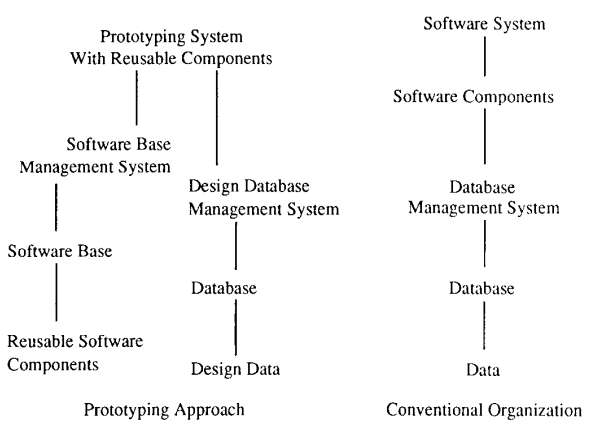
The requirements are firmed up iteratively in a rapid prototyping approach through the examination of executable prototypes as well as by negotiations between customer and designer. The designer constructs a prototype based on the requirements, and examines the execution of the prototype together with the customer. The requirements are adjusted based on feedback from the customer, and the prototype is modified accordingly until both the customer and the designer agree on the requirements. This process is illustrated below.



A prototype can also be used to specify a well modularized skeleton design for the intended system and to validate the important attributes of the intended system, e.g., timing constraints, input and output formats, or interfaces between modules. Rapid prototyping is a useful tool in feasibility studies. Prototypes of critical subsystems or difficult parts of a complicated system can significantly in-

crease the confidence that the system can be built before large amounts of effort and expense are committed to the project. Rapid prototyping helps in estimating costs, since the cost of the intended system is usually proportional to the cost of the prototype. The experiences gained in applying rapid prototyping to special applications, e.g., database design, metaprogramming method and others, have substantiated the expected cost relationships between the prototype and the completed system [9], [18].

Software tools are needed to make rapid prototyping practical. An initial description of a framework for a rapid prototyping environment based on reusability can be found in [30]. Since automatic program generation is not yet practical, reusing existing system components appears to be the most economic approach for constructing prototypes. Our approach depends on the prototype description language PSDL and a SOFTWARE BASE [27] containing a large set of reusable software components. The software base management system is responsible for organizing, retrieving, and instantiating reusable software components from the software base, while the design database is responsible for managing the versions and alternatives of the prototype design, as illustrated in the following diagram.



We assume that a software base management system supports the retrieval of the set of software components whose specifications match a given template. The reusable components in the software base are used to realize subsystems of the prototype, and the available reusable components are used to guide the decomposition process by which the behavior of the prototype is refined. PSDL is used to describe the connections between the components of a prototype, and to specify the behavior of the reusable components in the prototype as well as those in the software base. A powerful, easy to use, and portable prototype description language is a critical part of an automated rapid prototyping environment. Such a language is needed before the tools in the environment can be built.

B. Requirements for a Prototype Description Language

A language for supporting rapid prototyping of large real time systems has different requirements from a general purpose programming language or a specification language. In addition to being executable, the language must support the specification of requirements for the system and the functional description of the component modules. The module specification serves as the basis for organizing and retrieving the reusable components in a software base. Since rapid prototyping involves many design modifications, the language must make it easy for the system designer to create a prototype with a high degree of module independence [28], and to preserve its good modularity properties across many modifications. The prototype system description language has to be sufficiently easy to read to serve as design documentation, and also has to be formal enough for mechanical processing in the rapid prototyping environment.

The design of PSDL was motivated by the reasons mentioned above and by the requirements listed below:

1) PSDL should be based on a simple computational model that limits and exposes the interaction between system modules to encourage good modularizations of PSDL prototypes. An associated prototyping method for rapidly creating, modifying, and enhancing prototypes of large systems, which should be consistent with the language and to make the most efficient use of the language, should be designed based on the same computational model together with PSDL.

2) PSDL prototypes should be executable, so that the customer can observe the operation of the prototype.

3) PSDL should be simple and easy to use.

4) PSDL should support hierarchically structured prototypes, to simplify prototyping of large and complex systems. The PSDL descriptions at all levels of the designed prototype should be uniform.

5) PSDL should apply at both the specification and design levels to allow the designer to concentrate on designing the prototype without the distraction of transforming one notation into another.

6) PSDL should be suitable for specifying the retrieval of reusable modules from a software base, to avoid multiple specifications for each module.

7) PSDL should support both formal and informal module specification methods, to allow the designer to work in the style most appropriate to the problem.

8) PSDL should harmoniously support the basic concepts of data abstraction, function abstraction, and control abstraction to aid the construction of large prototypes.

9) PSDL should contain a set of abstractions suitable for constructing real-time systems.

After looking for an executable language to design prototypes based on a requirements set, we realized that our choices are mostly limited to programming languages. We are convinced that high level abstractions and brief and powerful language structures are definitely needed to simplify the design at a conceptual level. Many requirements specification and conceptual modeling languages are suitable at a high level, but unfortunately most of them are not executable. Some of the existing programming languages are too inflexible and too difficult to use. Many

kinds of coupling problems between modules of a system are not preventable in a programming language because conventional programming languages are required to execute very efficiently on conventional machines. Strong coupling can make a rapid prototyping effort fail because modifications get progressively more difficult and error prone, so conventional programming languages cannot be adequate for prototyping. Consequently the design of a special purpose language for rapid prototyping has to be considered.

We developed PSDL [23] because we could find no existing languages that met all of the above requirements.

C. Previous Work

Our work has been influenced by earlier work in four main areas: rapid prototyping, design languages, modeling of real-time systems, and reusable software.

Prototyping has become increasingly popular in system development. There is a popular branch of rapid prototyping work aimed at database applications [9]. The work is useful for prototyping business information systems, but it does not address real-time constraints or systems dominated by computation rather than data management. Several approaches are based on programming languages [15], [16]. These systems do not link very well to customer requirements, and do not address real time constraints. SREM [2] is a pioneering piece of work on the use of prototypes for validating requirements. This work addresses real time constraints, machine assisted generation of simulations, and tracing aspects of a simulation to customer requirements. However, SREM does not support abstractions very well and does not have a clearly defined computational model that limits interactions between modules. PAISLey [32] addresses the operational specification of real time systems. Operational specifications can be used for prototyping based on manually produced code. This work uses applicative functions to describe the state transition functions of cyclic processes. The hierarchical structure of a design cannot be captured in this approach because a cyclic process cannot be decomposed into more primitive cyclic processes. Control abstractions are not supported, and scheduling constraints are implicitly encoded in the way exchange functions are nested. Data abstractions are not supported either, making the approach cumbersome for prototyping large systems. Our work combines a clear interface-oriented computational model with data abstractions, operator abstractions, and control abstractions.

A prototyping language must have the characteristics of a good design language, because the structure of a prototype must be understandable and easy to modify. Early design languages [28], [19] were not executable, although more recent work has promise in this direction [8]. These languages do not support real time constraints or requirements tracing. Some design languages address the design and specification levels, but are not executable [4]. Other work on executable specifications [20] has taken the automatic transformation of specifications into running systems as a distant long term goal, and has concentrated on generating run-time checks from the specifications in the short run. These approaches are not sufficiently well developed to produce results applicable to rapid prototyping in the near future.

Some of the work on modeling real time systems has focused on the scheduling problems associated with real time constraints [25], [26]. These results are important for execution of PSDL prototypes. Languages for specifying real time systems have also been investigated [13], [19], [11].

Methods for enhancing the reusability of software [17], [31] are important for managing the software base [30], [27], which is one of the building blocks used in our work. A promising approach to making software components more flexible by the automated combination of different versions of their parts is reported in [7].

II. PSDL Concepts and Constructs

PSDL supports the prototyping of large systems by providing a simple computational model that is close to the designer's view of real time systems. The model is described in more detail below. PSDL supports operator, data, and control abstractions, and encourages hierarchical decompositions based on both data flow and control flow.

A. Computational Model

To provide a small and portable set of PSDL constructs with a clear semantics it is necessary to explore the mathematical model behind the language constructs. PSDL is based on a computational model containing OPERATORS that communicate via DATA STREAMS. Each data stream carries values of a fixed abstract data type [12]. Each data stream can also contain values of the built-in type EXCEPTION, as explained in Section II-A-4. The operators may be either data driven or periodic. Periodic operators have traditionally been the basis for most real time system design, while the importance of data driven operators for real-time systems is beginning to be recognized [24].

Formally the computational model is an augmented graph

$$G = (V, E, T(v), C(v))$$

where V is the set of vertices, E is the set of edges, $T(v)$ is the maximum execution time for each vertex v, and $C(v)$ is the set of control constraints for each vertex v. Each vertex is an operator and each edge is a data stream. The first three components of the graph are called the ENHANCED DATA FLOW DIAGRAM.

1) Operators: An operator is either a FUNCTION or a STATE MACHINE. When an operator fires, it reads one data object from each of its input streams, and writes at most one data object on each of its output streams. The output objects produced when a function fires depend only on the current set of input values. The output values produced when a state machine fires depend only on the current set of input values and the current values of a finite number

of internal STATE VARIABLES. We have found that operators of these two types are useful for prototyping real-time systems.

Operators are either ATOMIC or COMPOSITE. Atomic operators cannot be decomposed in terms of the PSDL computational model. Composite operators have realizations as data and control flow networks of lower level operators. If the output of an operator A is an input to another operator B, then there is an implicit precedence relationship between the two, which says that A must be scheduled to fire before B. A composite operator whose network contains cycles is a state machine. In such a case, one of the data streams in each cycle is designated as the state variable controlling the feedback loop, and an initial value is specified for each state variable. The state variables serve to break the circular precedence relationships among the operators which would otherwise be implied by the data flow relationships. In the example shown below, there is a cycle consisting of the streams S and Y. The stream X is the only input to the composite operator realized by the network, and Z is its only output.

In PSDL S would be designated as the state variable of this cycle by including a description of the form

states S initially S-0

in the specification part of the composite operator. S-0 represents an expression giving the initial value for S.

2) Data Streams: A data stream is a communication link connecting exactly two operators, the producer and the consumer. In Fig. 1, Y is a data stream from producer op-1 to consumer op-2. Communication links with more than two ends are realized using copy and merge operators. Each stream carries a sequence of data values. Streams have the pipeline property: if a and b are two data values in data stream Y and the data value a is generated by op-1 before the data value b is generated then it is impossible for a to be delivered to op-2 after b is delivered.

There are two types of data streams—DATA FLOW STREAMS and SAMPLED STREAMS. A data flow stream guarantees that none of the data values is lost or replicated, while a sampled stream does not make such a guarantee. A data flow stream can be thought of as a fifo queue, while a sampled stream can be thought of as a cell capable of containing just one value, which is updated whenever the producer generates a new value. Since real-time systems must often operate within a (small) bounded memory, the finite queue length imposes a restriction on the relative execution rates of two operators communicating via a data flow stream. A sampled stream imposes no such constraint, since it can deliver a value more than once if the consumer demands more values before the producer has provided a new value, and it can discard the previous value if the producer provides a new value before the consumer has used the previous one.

For example, suppose op-1 in the Fig. 1 produces a

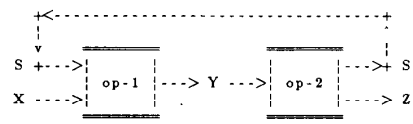


Fig. 1. A simple state machine.

sequence of data values $d1$, $d2$, $\cdots$, dn-1, dn in data stream Y. A queued sequence for Y is shown below.

$$dn\ dn\text{-}1\ \cdots\ d2\ d1$$

If Y is a data flow stream then op-2 consumes all of the data values in data stream Y in the order $d1$, $d2$, $\cdots$, dn-1, dn. If op-1 produces data values in data stream Y faster than op-2 consumes them, the length of the queued sequence increases until an overflow occurs. If op-2 consumes data values faster than op-1 produces them, op-2 has to wait until the queued sequence becomes nonempty. This implies that with a data flow stream op-2 may not meet its real time constraints unless its data rate closely matches that of op-1.

Suppose Y is a sampled stream instead, op-1 produces one data value in data stream Y every 2 ms, op-2 consumes one data value every 3 ms, and a size one buffer is used for containing the data values. In this case, op-1 produces data values at the following times as shown in Fig. 2, and op-2 consumes data values at different times as shown in Fig. 3.

Since the buffer contains only one data value at a given time, the sequences of data values in sampled stream Y consumed by op-2 starting at different initial times can be different. Op-2 consumes the values $d2$, $d3$, $d5$, $d6$, $\cdots$ with an initial time 0 and $d1$, $d2$, $d4$, $d5$, $\cdots$ with initial time 1, as shown in Fig. 3. If op-2 consumes one data value in the sampled stream every 1 ms, the consumed sequences of data values have repeated values, e.g., $d1$, $d1$, $d2$, $d2$, $d3$, $d3$, $\cdots$ or $d1$, $d2$, $d2$, $d3$, $d3$, $\cdots$.

Data flow streams must be used in cases where each data value represents a unique transaction or request that must be acted on exactly once. For example, the transactions in a system for electronic funds transfer would be transmitted along a data flow stream. Sampled data streams are often used for simulating continuous streams of information, where only the most recent information is meaningful. For example, an operator that periodically updates a software estimate of the system state based on sensor readings would use a sampled stream.

In PSDL the stream type is determined from the activation conditions for the consumer operator, rather than being explicitly declared. The rules for determining stream types are explained in the Section II-C-2.

3) Exceptions: PSDL exceptions are values of a built in abstract data type called EXCEPTION. This type has operations for creating an exception with a given name, for detecting whether a value is an exception with a given name, and for detecting whether a value is normal (i.e.,

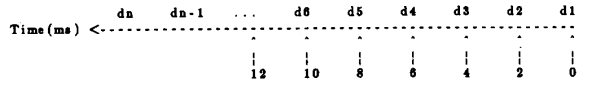


Fig. 2. Producer timing chart.

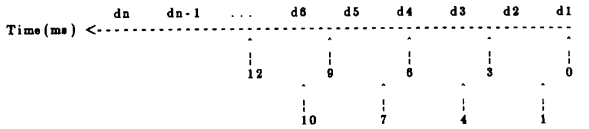


Fig. 3. Consumer timing chart.

belongs to some data type other than EXCEPTION). PSDL provides a shorthand syntax for the latter two operations, as illustrated in the following example of a PSDL predicate

x: overflow AND y: NORMAL

which is true if the input value x is the exception value with the name "overflow" and the input value y is normal, as indicated by the PSDL keyword "NORMAL." Predicates like this can be used to control the conditional execution of an operator or the conditional transmission of an output value.

Values of type EXCEPTION can be transmitted along data streams just like values of the normal type associated with the stream. Exceptions are encoded as data values in PSDL to decouple the transmission of an exceptional result from the scheduling of the actions for handling the exception, and to provide a programming language independent interface between atomic operators. This makes it possible to use atomic operators realized in several different programming languages in the same PSDL prototype.

B. Abstractions

Abstractions are an important means for controlling complexity [6], which is especially important in rapid prototyping because a system must appear to be simple to be built or analyzed quickly. PSDL supports three kinds of abstractions: data abstractions, operator abstractions, and control abstractions.

Data abstractions decouple the behavior of a data type from its representation. This is especially important in prototyping because the behavior of the intended system is only partially realized, capturing only those aspects important for the purposes of the prototype. The behavior of the prototype data is also a partial simulation of the data in the intended system, so that the data representations in the prototype and the intended system are likely to be quite different. Data abstraction allows the data interfaces to be described independently of the representation of the data, so that the interfaces for the operations on the data can be the same in the prototype and in the intended system. Aspects of the data not included in the prototype will be reflected in extra operations on the type, which appear in the intended system but not in the prototype. It is important to have common interfaces between the prototype and the intended system because it makes comparisons easier during the validation of the intended system, and because it enables the structure of the prototype design to be reused in the intended system where appropriate.

Control abstraction are important for simplifying the design of real-time systems, because much of the complexity of such systems lies in their control and scheduling aspects.

1) Operator Abstractions: An operator abstraction is either a functional abstraction or a state machine abstraction. Both functional and state machine abstractions are supported by the PSDL constructs for operator abstractions. PSDL operators have two major parts: the SPECIFICATION and the IMPLEMENTATION. The specification part contains attributes describing the form of the interface, the timing characteristics, and both formal and informal descriptions of the observable behavior of the operator. The attributes both specify the operator and form the basis for retrievals from a reusable component library or software base. The size and the content of the set of attributes may vary depending on the specific usage, underlying language, or the type of the modules specified. In our application, it consists of GENERIC PARAMETERS, INPUT, OUTPUT, STATES, EXCEPTIONS, and TIMING INFORMATION.

A PSDL operator corresponds to a state machine abstraction if its specification part contains a STATES declaration, and otherwise it corresponds to a functional abstraction. The STATES declaration gives the types of the state variables and their initial values. The state variables of a PSDL state machine are LOCAL, in the sense that they can be updated only from the inside of the machine. This restriction prevents coupling by means of shared state variables, and is one of the features of PSDL that leads to good modularizations. It is also important for making the correctness of distributed implementations independent of the number of processors.

The implementation part determines whether the operator is atomic or composite. Atomic operators have a keyword specifying the underlying programming language (Ada in our application), followed by the name of the implementation module implementing the operator. This name is filled in as the result of a successful retrieval from the software base, or is supplied by the designer in case the module cannot be constructed from reusable components and must be coded manually in a underlying programming language. Composite operators have the attributes COMMUNICATION GRAPH, INTERNAL DATA, CONTROL CONSTRAINTS, and INFORMAL DESCRIPTION.

2) Data Abstractions: All of the PSDL data types are immutable, so that there can be no implicit communication by means of side effects. Both mutable data types and global variables have been excluded from PSDL to help prevent coupling problems in large prototype systems. If

many modules communicate implicitly via a shared data structure or global variable, then it is easy to inadvertently interfere with a module by making an apparently unrelated change to another module. Repairing such faults is too time consuming to be tolerable in a rapid prototyping effort.

The PSDL data types include the immutable subset of the built-in types of Ada [1], user defined abstract types [12], the special types TIME and EXCEPTION, and the types that can be built using the immutable type constructors of PSDL. The PSDL type constructors (see Table I) were chosen to provide powerful data modeling facilities with a small set of semantically independent structures (cf. [5]).

Finite sets, sequences, and mappings correspond to the usual mathematical concepts. Tuples are finite Cartesian products, with operations for constructing tuples and for extracting components. One_ofs are tagged disjoint unions of a finite number of other types, with operations for constructing one_of values with a given tag (injections), for testing whether a one_of value has a given tag (domain predicates), and for extracting the data component of a one_of (projections). Relations are n-ary mathematical relations, with operations that are commonly used in relational databases (select, project, join, union, set difference, etc.).

Values of the special type TIME behave like integers, except that units may be specified (microseconds, milliseconds, seconds, minutes, or hours). Time values are assumed to be in milliseconds if the units are not given explicitly.

The basic construct of abstract data type in PSDL contains two parts: SPECIFICATION and IMPLEMENTATION. An example of a user defined abstract type in PSDL with details is given in Section III.

3) Control Abstractions: The control abstractions of PSDL are represented as enhanced data flow diagrams augmented by a set of control constraints. As a major property of real time systems, periodic execution is supported explicitly. Order of execution is only partially specified, and is determined from the data flow relations given in the enhanced data flow diagrams, based on the rule that an operator consuming a data value must not start until after the operator producing the data value has completed. This constraint applies only if the operators have the same period or if neither is periodic. If the order of execution for two operators is not determined by this rule, then both can run concurrently if sufficiently many processors are available. Conditional execution is supported by PSDL triggering conditions and conditional outputs (see Section II-C-4).

C. Control Constraints

The control aspect of a PSDL operator is specified implicitly, via control constraints, rather than giving an explicit control algorithm. There are several aspects to be specified: whether the operator is PERIODIC or SPORADIC, the triggering condition, and output guards. The stream types for the data streams in the enhanced data flow diagram are determined implicitly, based on the triggering conditions.

TABLE I
PSDL TYPE CONSTRUCTORS

set[item: type]
sequence[item: type]
map[from: type, to: type]
tuple[tag_1: T_1, ... , tag_n: T_n]
one_of[tag_1: T_1, ... , tag_n: T_n]
relation[tag_1: T_1, ... , tag_n: T_n]

1) Periodic and Sporadic Operators: PSDL supports both periodic and sporadic operators. Periodic operators are triggered by the scheduler at approximately regular time intervals. The scheduler has some leeway: a periodic operator must be scheduled to complete sometime between the beginning of each period and a deadline, which defaults to the end of the period. Sporadic operators are triggered by the arrival of new data values, possibly at irregular time intervals.

A PSDL operator is periodic if a period has been specified for it, and otherwise it is sporadic. A period can be specified explicitly, or it can be inherited from a higher level of decomposition in a hierarchical prototype (see Section II-E).

2) Data Triggers: Any PSDL operator can have a data trigger. There are two types of data triggers in PSDL, as illustrated by the following examples.

OPERATOR p TRIGGERED BY ALL x, y, z
OPERATOR q TRIGGERED BY SOME a, b

In the first example the operator p is ready to fire whenever new data values have arrived on all three of the input arcs x, y, and z. This rule is a slightly generalized form of the natural data flow firing rule [10], since in PSDL a proper subset of the input arcs can determine the triggering condition for an operator, without requiring new data on all of the input arcs. This kind of data trigger can be used to ensure that the output of the operator is always based on fresh data for all of the inputs in the list, and can be used to synchronize the processing of corresponding input values from a number of input streams.

In the second example, the operator q fires when any one of the inputs a and b gets a new value. This kind of activation condition guarantees that the output of operator q is based on the most recent values of the critical inputs a and b mentioned in the activation condition for q. If q has some other input c, the output of q can be based on old values of c, since q will not be triggered on a new value of c until after a new value for a or b arrives. This kind of trigger can be used to keep software estimates of sensor data up to date.

Every operator must have a period or a data trigger, or both. If a periodic operator has a data trigger, the operator is conditionally executed with the data trigger serving as an input guard. Conditionally executed operators are further discussed in the next subsection.

3) Timers: A timer is a special kind of abstract state machine whose behavior is similar to a stopwatch. Timers are used to record the length of time between events, or the length of time the system spends in a given state. This facility is needed to express relatively sophisticated aspects of real-time systems, such as timeouts and minimum refresh rates.

The state of a timer can be modeled as a time value and a boolean RUN switch. The value of the timer increases at a fixed rate reflecting the passage of real time when the RUN switch is on, and remains constant when the RUN switch is off.

There are four primitive operations for interacting with timers: READ, START, STOP, and RESET. The READ operation returns the current value of the timer without affecting the RUN switch. The START operation turns the RUN switch on without affecting the value of the timer. The STOP operation turns the RUN switch off without affecting the value of the timer. The RESET operation turns the RUN switch off and sets the value of the timer to zero.

Timers are treated specially in PSDL because they provide a nonlocal means of control. The PSDL declaration

TIMER *t*

creates an instance of the generic state machine described above, with the fixed name *t*. The name of a timer can be used like a PSDL input variable, whose value is the result of the READ operation of the timer. The value of a timer can be affected by PSDL control constraints of the forms

START TIMER *t*,
STOP TIMER *t*, and
RESET TIMER *t*.

These control constraints can appear anywhere the name *t* is visible, with the effect of invoking the START, STOP, and RESET operations of the abstract timer *t*. The name of a timer is subject to the following visibility rules:

1) A timer is visible in the module in which it is declared.

2) If a timer is visible in a composite module, it is also visible in the components of the composite.

These visibility rules allow only the module declaring a timer and its components at all levels to control the timer. The value of a timer can be used by modules outside the scope of the timer's name by being transmitted along a data stream. The timer itself cannot escape from the scope of its name by being transmitted along a data stream because it is a state machine rather than a data value.

4) Conditionals: PSDL supports two kinds of conditionals: conditional execution of an operator, and conditional transmission of an output. These constructs handle the controlled input and output of a PSDL operator.

a) Conditional Execution: PSDL operators can have a TRIGGERING CONDITION in addition to or instead of a data trigger. Two examples of operators with triggering conditions are shown below.

OPERATOR *r* TRIGGERED BY SOME *x*, *y*
IF *x*: NORMAL AND *y*: critical
OPERATOR *s* TRIGGERED IF *x*: critical

The first example shows the control constraints of an operator with both a data trigger and a triggering condition. The operator *r* fires only when one or both of the inputs *x* and *y* have fresh values, *x* is a normal data value, and *y* is an exceptional data value with the exception name CRITICAL.

The second example shows the control constraints of an operator *s* with a triggering condition but no data trigger. In this example *s* must be a periodic operator with an input *x* since sporadic operators must have data triggers, and triggering conditions can only depend on timers and locally available data. In this case the value of *x* is tested periodically to see if it is a CRITICAL exception, and the operator *s* is fired if that is the case. Both of these examples illustrate ways of using PSDL operators to serve as exception handlers.

In general, the triggering condition acts as a guard for the operator. If the predicate is satisfied, the operator fires and reads its inputs. If the predicate is not satisfied, the input values are read from the input data streams without firing the operator. If a periodic operator has a data trigger or a triggering condition, then the guard predicate is tested periodically, and the operator is fired on those periods where the guard is found to be true. The guard predicate of an operator can depend only on the input values to the operator and on the values of timers. The predicate can make use of the operators of the abstract data types carried by the input streams, allowing a structure similar to a guarded command, where different operators handle an input depending on some computable properties of the input value.

b) Conditional Output: An example of a control constraint specifying a conditional output is shown below.

OPERATOR *t* OUTPUT *z* IF $1 < z$ AND $z <$ max

The example shows an operator with an output guard, which depends on the input value MAX and the output value *z*.

In general an output guard acts as if the corresponding unconditional output had been passed through a conditionally executed filtering operator with the same predicate as a triggering condition. The filtering operator passes the input value to the output stream unchanged, provided that the predicate evaluates to TRUE. If the predicate evaluates to FALSE, the filter removes the value from its input stream without affecting its output stream. An output predicate can depend only on the input values to the operator, the output values of the operator, and values of timers.

Output guards are convenient but they do not strictly increase the expressive power of the language, since they can be simulated by adding an explicit filter operator, at the cost of some additional output streams to the original

operator (since the output guard can depend on the INPUTS of an operator as well as on its outputs).

5) Producing and Handling Exceptions: Exceptions can be produced in both PSDL and the underlying programming language. If a reusable component causes an exception that is not handled within the component, the PSDL execution support system must catch the exception and convert it to a PSDL exception. The resulting exception value is transmitted on all of the output streams of the component, subject to any output guards the component may have. Exceptions can also be produced by PSDL control constraints. For example, the control constraint

OPERATOR *f* EXCEPTION *e* IF $x > 100$

transmits the exception value named *e* on all of the output streams of *f* instead of the values actually computed by *f* whenever the input value *x* is greater than 100.

Exceptions can be handled in the underlying programming language, or they can be suppressed using PSDL. If a reusable component receives a PSDL exception value on one of its input streams then the PSDL execution support system must raise the corresponding exception of the underlying programming language. In such a case the module must handle the exception using the exception handling mechanism of the underlying programming language.

Suppose the operator op-1 produces a data stream *Y* and the operator op-2 consumes it. Exceptions can be suppressed either by a PSDL output guard of the form

OPERATOR op-1 OUTPUT *Y* IF *Y*: normal

or a PSDL input guard of the form

OPERATOR op-2 TRIGGERED IF *Y*: normal.

In both cases op-2 will not receive any exceptional input values.

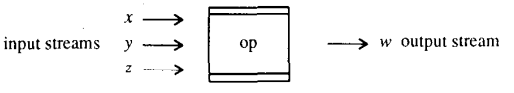
6) Derivation of Stream Types: The data trigger of an operator determines the stream types of its input streams by the following rules.

1) If a stream is listed in an ALL data trigger, then it is a data flow stream.

2) All streams not constrained by the first rule are sampled streams.

These rules are motivated by the fact that an operator must be executable whenever its triggering conditions are satisfied. In particular, values of streams that are not mentioned at all, or are mentioned in SOME DATA triggers can be demanded at arbitrary times, which is inconsistent with the fact that data flow streams cannot allow the consumer to read more values than the producer has written. Consequently rule 1 captures the most general situation where data flow streams make sense.

For example, suppose operator op has the input streams *x*, *y*, *z*, and the output stream *w*.



Under the following control constraint

OPERATOR op TRIGGERED BY ALL *x*, *y*

x, *y* are data flow streams while *z* is a sampled stream. Under a different control constraint

OPERATOR op TRIGGERED BY SOME *x*, *y*

x, *y*, *z* are all sampled streams. In either case, the stream type of *w* is not affected by the control constraint associated with its producer operator op.

These rules interact with the inheritance of properties for hierarchically defined operators, as described in Section II-E.

D. Timing Constraints

Timing constraints are an essential part of specifying real time systems. The most basic timing constraints are given in the specification part of a PSDL module, and consist of the MAXIMUM EXECUTION TIME, the MAXIMUM RESPONSE TIME, and the MINIMUM CALLING PERIOD. The maximum execution time is an upper bound on the length of time between the instant when a module begins execution and the instant when it completes. The maximum execution time is a constraint on the implementation of a single module, and does not depend on the context in which the module is used. This constraint may apply to all PSDL operators.

The last two constraints are important for sporadic operators. The maximum response time for a sporadic operator is an upper bound on the time between the arrival of a new data value (or set of data values for operators with the natural data flow firing rule) and the time when the last value is put into the output streams of the operator in response to the arrival of the new data value. The maximum response time for a periodic operator is an upper bound on the time between the beginning of a period and the time when the last value is put into the output streams of that operator during that period. The maximum response time includes potential scheduling delays, while the maximum execution time does not.

The minimum calling period is a constraint on the environment of a sporadic operator, consisting of a lower bound on the delay between the arrival of one set of inputs and the arrival of the next set. In a PSDL specification every sporadic operator with a maximum response time constraint must have a corresponding minimum calling period constraint.

More complicated timing constraints can be given in the implementation part, using event controlled timers, triggering conditions, and output conditions. The use of event controlled timers is illustrated by the example in Section III-C.

E. Hierarchical Constraints

PSDL operators are defined in a hierarchical structure. There are a number of constraints associated with the hierarchical structure. The most fundamental constraints concern locality of data streams. If a composite operator

is realized as a network of component operators, then an input stream of a component operator must be one of the input streams of the composite operator if the stream is produced by an operator outside the composite operator. Similarly every output stream of a component operator must also be an output stream of the composite operator if it is consumed by an operator outside the composite operator. Furthermore, each input of a composite operator must be an input of at least one of its components, and each output of the composite operator must be an output of at least one of its components.

As explained above, stream types are derived from the triggering conditions of the consumer operator. If the consumer is a composite operator, then at least one of its components is another consumer for the same stream. Both the composite consumer and its component consumer induce constraints on the stream type.

PSDL timing constraints also impose some consistency requirements between the various levels of a hierarchical design. The maximum execution time and the maximum response time of a subnetwork must be no larger than those of the composite operators realized by the subnetwork.

Other constraints associated with the hierarchy are best described in terms of inheritance rules. The components of a composite operator inherit the following properties from the composite:

1) The period.
2) The minimum calling period.
3) The stream types of the streams crossing the boundary of the composite.

A composite operator inherits the exceptions it produces from the component operators producing the output streams of the composite. This bottom-up inheritance rule requires a second pass over a design developed by a top-down methodology, to check that all of the exceptions have been either handled or included at the higher levels of abstraction. The PSDL execution support system should check all of the above constraints and produce error messages if they are not satisfied.

III. An Example of Prototyping Using PSDL

This section illustrates the usage of PSDL by means of an example, which is a simple system for treating brain tumors using hyperthermia. The treatment consists of inserting an antenna into the patient's brain, and using microwaves to heat the tumor to 42.5 degree C, killing the tumor cells but not the healthy cells. The purpose of the software system is to control the microwave power to maintain the temperature at the required levels, without endangering the patient by allowing the temperature to go too high or by taking too long to achieve the hyperthermia temperature.

The system uses feedback from temperature sensors in the antenna to monitor its progress. The use of sensors and a digital control to maintain some operating condition in an external system makes this example typical of many real time applications. Large applications differ from the example only in degree, containing more details, and also more independent systems that must be controlled simultaneously. The presence of many simultaneous processes makes the scheduling problem more complicated, but does not qualitatively alter the way in which PSDL is used by the designer.

The prototype in the example discussed below has three levels, each of which is presented in a separate subsection. The lowest level components have specifications in PSDL (given in Section III-D) and implementations in Ada (given in Section III-E).

A. Requirements for the Hyperthermia System

A set of requirements for our example are shown below. PSDL assumes that the requirements are structured as a set of named items. The PSDL facility for recording the correspondence between the requirements and the parts of the prototype works best if each item in the requirements represents a single constraint and different items represent independent constraints. The requirements can be given in English as in the example below, or a more formal notation can be used (cf. [14]).

1) Shutdown: Microwave power must drop to zero within 300 ms of turning off the treatment switch.

2) Temperature Tolerance: After the system stabilizes, the temperature must be kept between 42.4 and 42.6 degrees C.

3) Maximum Temperature: The temperature must never exceed 42.6 degrees C.

4) Startup Time: The system must stabilize within 5 minutes of turning on the treatment switch.

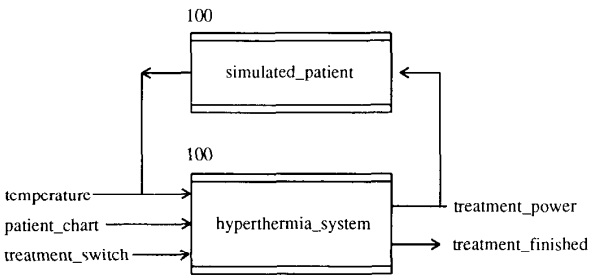
5) Treatment Time: The system must shut down automatically when the temperature has been above 42.4 degrees C for 45 minutes.

B. First Level

The construction of a prototype proceeds top down, with a number of levels of refinement. The refinements are based on functional decompositions given by the associated methodology for rapid prototyping [22]. A PSDL description for the highest level of the brain tumor treatment system is shown below.

```
OPERATOR brain_tumor_treatment_system
SPECIFICATION
  INPUT patient_chart: medical_history,
    treatment_switch: boolean
  OUTPUT treatment_finished: boolean
  STATES temperature: real
    INITIALLY 37.0
  DESCRIPTION
  { The brain tumor treatment system kills tumor cells
    by means of hyperthermia induced by micro-
    waves.
  }
END
```

```
IMPLEMENTATION
  GRAPH
```




```
  DATA STREAM treatment_power: real

  CONTROL CONSTRAINTS
    OPERATOR hyperthermia_system
      PERIOD 200 BY REQUIREMENTS shut-
        down
    OPERATOR simulated_patient
      PERIOD 200
  DESCRIPTION { paraphraser output }
END
```

The brain tumor treatment system is described as a periodically executing feedback loop, which implements a state machine. The period is chosen to meet the emergency shutdown requirement, which requires the system to set the power to zero within 300 ms of the time the treatment switch is turned off. This requirement will be met if the sum of the period and the maximum execution time of the hyperthermia system do not exceed 300 ms. Since the treatment switch can change at an unpredictable moment, it can be almost a full period before the system samples the value of the switch. It can take up to 100 ms more for the hyperthermia system to be executed and respond to the changed input signal. A tighter bound cannot be established without looking inside the hyperthermia system, which we want to avoid to preserve the hierarchical structure of the prototype. The timing estimates given above are approximations to the actual times. The accuracy of these values can be improved by measuring the running time of the prototype or by using a static timing analysis tool, which calculates worst case time bounds for loop free code using instruction times based on a particular compiler and machine.

The period must also allow the system to make adjustments to the power level fast enough to guarantee that the temperature remains in the allowable range. The correspondence between temperature tolerances and required response times is difficult to determine *a priori*, and would be determined in practice by means of experiments using the prototype. These experiments are likely to spark changes to the timing requirements as well as to the control algorithms. An important reason for building prototypes is to help determine the exact timing requirements that will suffice to guarantee functional properties of the system such as the temperature tolerance requirement.

A software simulation of the patient (together with the microwave generator, antenna, and temperature sensors) is included to allow the prototype to be tested. This is typical of many real-time applications, where the actual environment of the intended system is too dangerous or too expensive to risk while testing a prototype with unknown and possibly faulty properties. We believe that simulations of the environment of the software system are an essential part of rapid prototyping, and that any language for prototyping real time systems must support the construction of such simulations.

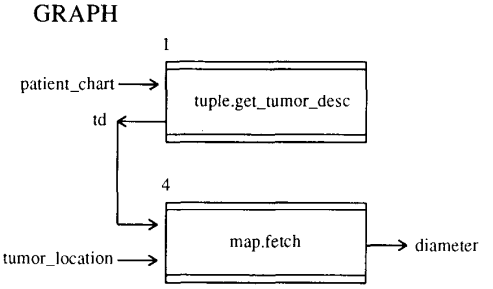
The brain tumor treatment system is specified in terms of the medical_history, an abstract data type which is used as one of the external inputs to the system. A partial PSDL description for this data type is given below. The complete data type has many other operations, but only those related to the brain tumor treatment system are given here.

```
TYPE medical_history
SPECIFICATION
  OPERATOR get_tumor_diameter
    SPECIFICATION
      INPUTS patient_chart: medical_history,
        tumor_location: string
      OUTPUTS diameter: real
      EXCEPTIONS no_tumor
      MAXIMUM EXECUTION TIME 5 ms
      DESCRIPTION
      { Returns the diameter of the tumor at a given
        location, produces an exception if no tumor
        at that location.
      }
  END
  . . .

  KEYWORDS patient_charts, medical_records,
    treatment_records, lab_records
  DESCRIPTION
  { The medical history contains all of the disease and
    treatment information for one patient. The opera-
    tions for adding and retrieving information not
    need by the hyperthermia system are not shown
    here.
  }
END

IMPLEMENTATION
  tuple[tumor_desc: map[from: string, to: real], . . . ]

  OPERATOR get_tumor_diameter
  IMPLEMENTATION
    GRAPH
```



```
    DATA STREAM td: tumor_description
    CONTROL CONSTRAINTS
      OPERATOR map.fetch
        EXCEPTION no_tumor
        IF not(map.has (tumor_location, td))
  END
  . . .
END
```

The data types TUPLE and MAP are built into PSDL. A tuple is the Cartesian product of a number of component types, with symbolic names for the components. A map is a function from a finite subset of one data type to another data type, and is similar to a lookup table. The fetch and get_tumor_description functions are primitive operations of the map and tuple types, so that they need not be refined any further. Note that tuple is a parameterized family of types with a get_X operation for each component name X. Note the use of the exception control constraint to turn an exception of the built-in map type into a different exception meaningful for the medical_history type.

C. Second Level

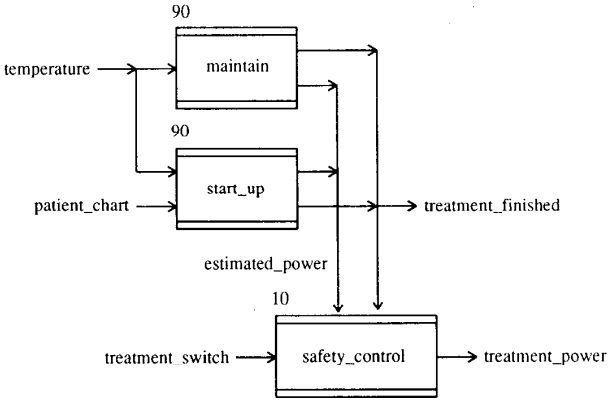
The PSDL description for the hyperthermia system is shown below. The PSDL description for the simulated patient is not shown, since the thermal properties of the human brain are not essential for this paper.

```
OPERATOR hyperthermia system

SPECIFICATION
  INPUT temperature: real,
    patient_chart: medical_history,
    treatment_switch: boolean
  OUTPUT treatment_power: real,
    treatment_finished: boolean
  MAXIMUM EXECUTION TIME 100 ms
    BY REQUIREMENTS temperature_tolerance
  MAXIMUM RESPONSE TIME 300 ms
    BY REQUIREMENTS shutdown

  KEYWORDS medical_equipment, temperature_control, hyperthermia, brain_tumors
  DESCRIPTION
  { After the doctor turns on the treatment switch, the
    hyperthermia system reads the patient's medical
    record, and turns on the microwave generator to
    heat the tumor in the patient's brain. The system
    controls the power level to maintain the hyperthermia temperature (42.5 degrees C.) for 45 minutes to kill the tumor cells. When the treatment is
    over the system turns off the power and notifies
    the doctor.
  }
END
```

```
IMPLEMENTATION
  GRAPH
```




```
  DATA STREAM estimated_power: real
  TIMER treatment_time

  CONTROL CONSTRAINTS
    OPERATOR start_up
      TRIGGERED IF temperature < 42.4
        BY REQUIREMENTS maximum_temperature
      STOP TIMER treatment time
      RESET TIMER treatment_time
        IF temperature < = 37.0

    OPERATOR maintain
      TRIGGERED IF temperature > = 42.4
        BY REQUIREMENTS maximum_temperature
      START TIMER treatment_time
        BY REQUIREMENTS treatment_time,
          temperature_tolerance
      OUTPUT treatment_finished
        IF treatment_time > = 45 min
        BY REQUIREMENTS treatment_time
  END
```

This example illustrates the use of an event controlled timer, a conditional output, and conditionally activated operators. The treatment_time timer is reset (to zero) whenever the temperature drops below body temperature (i.e., at the end of a treatment session). The timer is (re)started if the temperature is in the range for effective hyperthermia, and it is stopped if the temperature goes out of the range. The treatment_time timer is used to record the treatment time, and to control the transmission of the output treatment_finished from the MAINTAIN operator.

The MAINTAIN operator always produces the value TRUE for the treatment_finished switch, while the START_UP operator always produces the value FALSE for the treatment_finished switch. Since the output of MAINTAIN is conditional, the TRUE value is transmitted only when the predicate giving the output condition is true. The initial FALSE value persists until the conditional output is transmitted because the treat-

ment_finished is a sampled data stream (as are all of the other data streams in this example). Both START_UP and MAINTAIN are triggered conditionally. The guard predicates of these two operators are mutually exclusive, so that only one of the two is executed in any given period.

The treatment_finished signal is a output from the system, which informs the doctor that the treatment is over by means of an indicator light on the display panel. The treatment_finished signal is also used by the SAFETY_CONTROL to determine when the microwave power should be shut off.

D. Third Level

The specifications for START_UP, MAINTAIN, and SAFETY_CONTROL are given below. All three of these functions are leaf nodes in the decomposition tree, and are implemented as Ada modules.

```
OPERATOR start-up
SPECIFICATION
  INPUT patient_chart: medical_history,
    temperature: real
  OUTPUT estimated_power: real,
    treatment_finished: boolean
     BY REQUIREMENTS startup_time
  MAXIMUM EXECUTION TIME 90 ms
     BY REQUIREMENTS temperature_tolerance
  DESCRIPTION
  { Extracts the tumor diameter from the medical history and uses it to calculate the maximum safe treatment power. Estimated power is zero if no tumor is present. Treatment finished is true only if no tumor is present.
  }
END

IMPLEMENTATION Ada start_up
END

OPERATOR maintain
SPECIFICATION
  INPUT temperature: real
  OUTPUT estimated_power: real,
    treatment_finished: boolean
  MAXIMUM EXECUTION TIME 90ms
     BY REQUIREMENTS temperature_tolerance
  DESCRIPTION
  { The power is controlled to keep the power between 42.4 and 42.6 degrees C.
  }
END

IMPLEMENTATION Ada maintain
END

OPERATOR safety_control
SPECIFICATION
  INPUT treatment_switch, treatment_finished: boolean, estimated_power: real
  OUTPUT treatment_power: real
     BY REQUIREMENTS shutdown
  MAXIMUM EXECUTION TIME 10 ms
     BY REQUIREMENTS temperature_tolerance
  DESCRIPTION
  { The treatment power is equal to the estimated power if the treatment switch is true and treatment finished is false, and otherwise the treatment power is zero.
  }
END

IMPLEMENTATION Ada start_up
END
```

E. Ada Modules

Ada implementations of the modules specified in the previous section are shown below.

```
WITH medical_history_package;
  USE medical_history_package;
PROCEDURE start_up(
  patient_chart: IN medical_history;
      temperature: IN real;
      estimated_power: OUT real;
      treatment_finished: OUT boolean ) IS

  diameter: real;
  k: constant real := 0.5;
BEGIN
  diameter := get_tumor_diameter(
    patient_chart, ''brain_tumor'');
  estimated_power := k * diameter ** 2;
  treatment_finished := false;
EXCEPTION
  WHEN no_tumor =>
    estimated_power := 0.0;
    treatment_finished := true;
END start_up;
```

Note the PSDL exception that may result from the get_tumor_diameter operation has been treated as an Ada exception. It is the responsibility of the PSDL execution support system to map PSDL exceptions into the exceptions of the underlying programming language and back again whenever an exception crosses the boundary between a PSDL module and a module from the software base.

```
PROCEDURE maintain(temperature: IN real;
      estimated_power: OUT real;
      treatment_finished: OUT boolean ) IS

  c: CONSTANT real := 10.0;
BEGIN
  IF temperature > 42.5
    THEN estimated_power := 0.0;
  ELSE estimated_power := c * (42.5 - temperature);
  END IF;
  treatment_finished := true;
END maintain;
```

```
PROCEDURE safety_control(
  estimated_power: IN real;
      treatment_finished: IN boolean;
      treatment_switch: IN boolean;
      treatment_power: OUT real) IS
BEGIN
   IF treatment_switch and not treatment_finished
   THEN treatment_power := estimated_power;
   ELSE treatment_power := 0.0;
   END IF;
END safety-control;
```

All of the above modules are too small for further decomposition to be useful, and would be hand coded in the underlying programming language if they cannot be retrieved from the software base.

IV. Conclusion

The rapid construction of a prototype in PSDL is made possible by the associated methodology and support environment. The methodology relies on an improved modularization technique and reusable software components. The support environment reduces the efforts of the designer by automating some of the tasks involved in prototype construction. The most important aspects of the support environment are the prototype execution facilities, the software base, and the design entry facilities. Our approach is made possible by the Prototype System Description Language, which was designed to serve both as a conceptual tool for modeling real-time systems and as the link to the reusable components in the software base.

A. Prototyping Methodology

The main goals of the methodology associated with PSDL are to construct a prototype with a high degree of module independence [28] and to do so rapidly. The first goal is addressed by an improved modularization technique and a hierarchical approach as described below. The second goal is addressed by the support environment discussed in the next section. The PSDL prototyping methodology is discussed in more detail in [22], [23]. Module indpendence has a particular importance in our methodology because of the need for multiple modifications to the prototype. Since module independence is desirable for production software as well as for the prototype, we expect the modularization (but not necessarily the code) of the prototype to be used in the production system, although it may have to be refined in places. The code of the prototype usually cannot be used in the production system because the prototype is not a complete representation of the final system.

B. Support Environment

PSDL was designed to support an automated environment for rapid prototyping. The most important parts of the environment are a static scheduler, a translator, a dynamic scheduler, a software base management system, a syntax directed editor, and a paraphraser. The static scheduler, translator, and dynamic scheduler are discussed in [21]. Our strategy for implementing PSDL is to map the PSDL constructs into an underlying programming language. Most programming languages can be used for this purpose. Ada [1] is a particularly good candidate because of its features for large scale programming, notably packages and generic modules.

Reusable software is an important part of our methodology. PSDL serves as the link between the designer and the software base, which contains reusable components. We assume that each of the components in the software base contains a PSDL module specification as well as an implementation in the underlying programming language. When the designer gives a decomposition for a composite module, the behavior of each part of the composite module is given in the form of a PSDL module specification. The module specification is used as template for retrieval from the software base. Further decomposition of a module is attempted only if the retrieval from the software base fails. The software base management system does these retrievals based on approximate matching of the template and the specification parts of the reusable components in the software base. One kind of inexact match we expect to be important is the instantiation of a generic module in the software base, with actual generic parameters automatically bound in a way that provides an exact match to the template. This process is similar to a limited kind of unification. Analogs to resolution (cf. Prolog) may be useful in a more powerful facility for satisfying a retrieval request by a composite of several modules occurring in the software base. Effectively implementing retrievals based on approximate matching is a hard problem partially because it depends on the exact attributes chosen for the reusable modules in the underlying programming language, the organization of reusable modules in the software base, and the detailed architecture of software base management system [27]. PSDL has been designed to support a variety of retrieval schemes by providing a multidimensional form for specifications, consisting of a number of different attributes.

A syntax directed editor for PSDL can provide a user-friendly interface and significantly reduce the amount of effort required of the designer. A graphical interface with a high resolution display and a pointer device such as a mouse is convenient for manipulating the enhanced data flow diagram in the PSDL implementation of a composite module. The enhanced data flow diagram with control constraints can be translated into the GIST specification language [3]. A comparison of the output of the GIST paraphraser and the PSDL specification of the composite module can help to attain a correct implementation. In other words, the formal and informal descriptions for the functionality of the module in the SPECIFICATION part of a PSDL component and the output of the paraphraser in the IMPLEMENTATION part of PSDL component provide a useful redundancy for component description of a rapid prototype.

C. Prototype Execution

PSDL prototypes are executable if all required information is supplied, and the software base contains implementations for all atomic operators and types. To simplify the description of the PSDL translator we will assume here that Ada [1] is used for implementing both the reusable components in the software base and the PSDL execution support environment. The PSDL execution support system contains a static scheduler, a translator, and a dynamic scheduler. The static scheduler attempts to find a static schedule for the operators with real time constraints. The translator augments the implementations of the atomic operators and types with code realizing the data streams and activation conditions, resulting in a program in the underlying programming language that can be compiled and executed. Execution is under the control of a dynamic scheduler, which schedules the operators without real-time constraints and provides facilities for debugging and gathering statistics. More details can be found in [23].

D. PSDL

The PSDL language supports rapid prototyping of large real-time systems. The prototyping process is speeded up by automation and by reducing the conceptual burden of the analyst and prototype designer. PSDL was designed to interface to an automated support environment with a software base containing reusable software components. The language is small and simple, reducing the number of errors made by the designer and speeding up the prototyping process. It also helps the designer to organize and understand a complex prototype by eliminating some forms of tight coupling and by supporting abstractions and hierarchical decompositions. The constructs and abstractions supported by the language are useful for modeling real time systems and controlling the complexity of large systems. PSDL has constructs for recording timing constraints, for defining operator and data abstractions, and for specifying nonprocedural control constraints. The control constraints are a form of control abstraction that can be used to simplify the description of a real-time system at all levels of the abstraction hierarchy.

PSDL has been designed in parallel with a prototyping methodology, to make the two compatible. The methodology supports the best known modularization techniques, unifying the data flow and control flow decomposition criteria. A systematic methodology is essential for creating a prototype of a large system and keeping it under control. The facilities for tracing components of the prototype to particular requirements help in adjusting the prototype to fit modified requirements. The explicit and loosely coupled interfaces of PSDL help to localize the effects of prototype modifications. PSDL is useful for analyzing the potential benefit of multiple processors in real time systems because the description of the prototype's behavior is insensitive to the number of processors available.

The most important aspects of real time systems design are maximum response time and data synchronization. PSDL handles both of these aspects well. The PSDL execution support system should partially check the feasibility of a set of real time constraints, and additional confidence can be gained by metering the execution of the prototype. The data flow streams of PSDL provide a simple and abstract way of specifying and realizing data synchronization requirements. The sampled streams and sporadic operators of PSDL provide a convenient means for modeling interactions with external hardware components, which is another common problem in real-time systems design.

Promising directions for future research relevant to PSDL include better techniques for retrieving reusable software components from a software base, and computer aided techniques for partitioning computations with tight real-time constraints to run on distributed processors.

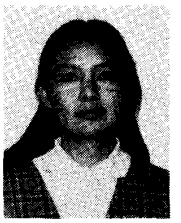
ACKNOWLEDGMENT

We would like to thank E. Frankowski, A. K. Mok, N. Roussopoulos, T. T. Song, and many colleagues at International Software Systems, Inc. for their kind help and their useful suggestions.

REFERENCES

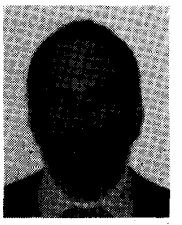
[1] *Ada Programming Language*, U. S. Dep. Defense, ANSI/MIL-STD-1815A, 1983.
[2] A. W. Alford, "A requirements engineering methodology for real-time processing requirements," *IEEE Trans. Software Eng.*, vol. SE-3, no. 1, pp. 60-68, Jan. 1977.
[3] R. Balzer, D. Cohen, and W. Swartout, "Tools for specification validation and understanding," Rome Air Development Center, Rep. RADC-TR-83-292, Dec. 1983.
[4] F. W. Beichter, O. Herzog, and H. Petzsch, "SLAN-4: A software specification and design language," *IEEE Trans. Software Eng.*, vol. SE-10, no. 2, pp. 155-162, Mar. 1984.
[5] V. Berzins and M. Gray, "Analysis and design in MSG.84: Formalizing functional specifications," *IEEE Trans. Software Eng.*, vol. SE-11, no. 8, pp. 657-670, Aug. 1985.
[6] V. Berzins, M. Gray, and D. Naumann, "Abstraction-based software development," *Commun. ACM*, vol. 29, no. 5, pp. 402-415, May 1985.
[7] V. Berzins, *On Merging Software Extensions, Acta Informatica*, vol. 23, fasc. 6, pp. 607-619, Nov. 1986.
[8] T. Cheatham, J. Townley, and G. Holloway, "A system for program refinement," in *Interactive Programming Environments*. New York: McGraw-Hill, 1984, pp. 198-214.
[9] J. Connell and L. Brice, "Rapid prototyping," Datamation, pp. 93-100, Aug. 1984.
[10] J. B. Dennis, G. A. Boughton, and C. K. C. Leung, "Building blocks for dataflow prototypes," in *Proc. Seventh Symp. Computer Architecture*, La Baule, France, May 1980.
[11] A. A. Faustini and E. B. Lewis, "Toward a real-time dataflow language," *IEEE Software*, vol. 3, no. 1, pp. 29-35, Jan. 1986.
[12] J. V. Guttag, E. Horowitz, and D. R. Musser, "Abstract data types and software validation," *Commun. ACM*, vol. 21, no. 12, pp. 1048-1064, Dec. 1978.
[13] V. H. Haase, "Real-time behavior of programs," *IEEE Trans. Software Eng.*, vol. SE-7, no. 5, pp. 494-501, Sept. 1981.
[14] K. L. Heninger, "Specifying software requirements for complex sys-

tems: New techniques and their applications," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 2-12, Jan. 1980.
[15] E. L. Ivie, "The programmer's workbench—A machine for software development," *Commun. ACM*, vol. 20, no. 10, pp. 746-753, Oct. 1977.
[16] P. Kruchten, E. Schonberg, and J. Schwartz, "Software prototyping using the SETL programming language," *IEEE Software*, vol. 1, no. 4, pp. 66-75, Oct. 1984.
[17] B. Leavenworth, "ADAPT: A tool for the design of reusable software," IBM Thomas J. Watson Research Center, Yorktown Heights, NY, Rep. RC 9728, 1982.
[18] L. S. Levy, "A metaprogramming method and its economic justification," *IEEE Trans. Software Eng.*, vol. SE-12, no. 2, pp. 272-277, Feb. 1986.
[19] G. Luckenbaugh, "The activity list: A design construct for real-time systems," M.S. thesis, Dep. Comput. Sci., Univ. Maryland, 1984.
[20] D. Luckham and F. W. von Henke, "An overview of Anna, a specification language for Ada," *IEEE Software*, vol. 2, no. 2, pp. 9-22, Mar. 1985.
[21] Luqi and V. Berzins, "Execution aspects of prototypes in PSDL," TR-86-2, Univ. Minnesota, Tech. Rep. TR-86-2, 1986.
[22] ——, "The rapid construction of PSDL prototypes," *IEEE Software*, Sept. 1988.
[23] Luqi, "Rapid prototyping for large software system design," Ph.D. dissertation, Dep. Comput. Sci., Univ. Minnesota, Minneapolis, 1986.
[24] L. MacLaren, "Evolving toward Ada in real time systems," in *Proc. ACM SIGPLAN Symp. Ada Programming Language SIGPLAN Notices*, pp. 146-155, Nov. 1980.
[25] A. K. Mok, "The design of real-time programming systems based on process models," in *Proc. IEEE 1984 Real Time Systems Symp.*, Austin, TX, Dec. 1984, pp. 5-17.
[26] ——, "The decomposition of real-time system requirements into process models," in *Proc. IEEE 1984 Real Time Systems Symp.*, Austin TX, Dec. 1984, pp. 125-133.
[27] N. Roussopoulos, "Architectural design of the SBMS," Dep. Comput. Sci., Univ. Maryland, Quarterly Rep. for the STARS SB/SBMS Project, Apr. 1985.
[28] W. Stevens, G. Meyers, and L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115-139, May 1974.
[29] R. T. Yeh, "Software engineering," *IEEE Spectrum*, pp. 91-94, Nov. 1983.
[30] R. T. Yeh, R. Mittermeir, N. Roussopoulos, and J. Reed, "A programming environment framework based on reusability," in *Proc. Int. Conf. Data Engineering*, Apr. 1984.
[31] R. T. Yeh, N. Roussopoulos, and B. Chu, "Management of reusable software," in *Proc. COMPCON*, Sept. 1984.
[32] P. Zave, "An operational approach to requirements specifications for embedded systems," *IEEE Trans. Software Eng.*, vol. SE-8, no. 3, pp. 250-269, 1982.



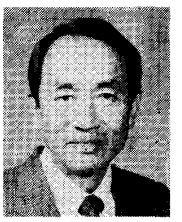
Luqi received the B.S. degree from Jilin University, China, and the M.S. and Ph.D. degrees in computer science from University of Minnesota, Minneapolis.

She worked on software research, development, and maintenance for the Science Academy of China, the Computer Center at University of Minnesota, and International Software Systems, and is currently an Assistant Professor at the Naval Postgraduate School and an adjunct assistant professor at the University of Minnesota. Her research interests include rapid prototyping, real-time systems, and software development tools.



Valdis Berzins (S'76-M'78) received the B.S., M.S., E.E., and Ph.D. degrees from Massachusetts Institute of Technology, Cambridge.

He is an Associate Professor of Computer Science at the University of Minnesota, Minneapolis, and the Naval Postgraduate School, Monterey, CA. He has taught at the University of Texas. His interests include software engineering and computer-aided design. His recent work includes papers on software merging, VLSI design, specification languages, and engineering databases. He has developed a number of specification languages and software tools.



Raymond T. Yeh (S'64-M'72-SM'78-F'83) is founder of International Software Systems, Inc. and SYSCORP International, Inc. Previously, he was Chairman of the Department of Computer Science at the University of Texas at Austin (1975-1978) and the Department of Computer Science at the University of Maryland at College Park (1979-1982). He was also Director of the Information Sciences Research Center at Maryland.

Dr. Yeh has played a major role in Software Engineering activities in the U.S. He was the founding Editor-in-Chief of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. He also founded the Technical Committee on Software Engineering of the IEEE Computer Society in 1974 and established the International Conference on Software Engineering (ICSE) in 1975. He has published 8 books and over 100 scientific articles. He has received many honors, including the IEEE Centenial Medal (1984), the CDC endowed chair of distinguished professor (1983), Honorary Professor of four major universities in China, Honorary Research Fellow of Fujitsu (Japan), and an award for outstanding achievements from the University of Maryland.